

ENTERPRISE CONNECTIVITY TO HANDHELD DEVICES

Related Application

115B
5 81 [This application is a continuation of U.S. Application No. 08/665,422 filed June 18, 1996.

Background of the Invention**Field of the Invention**

10 The present invention generally relates to client/server technology and, more specifically, to a client/server architecture for occasional connections between mobile computing devices and enterprise computing systems.

Background

15 In the current persistent connection client/server model, personal computer clients "connect" to a server on the network and request data from the server as needed by an application. This is usually performed by use of SQL (Structured Query Language). The connection between the client and server exists the entire time the application is in use, usually for hours at a time. This is not possible in a mobile model, because it is not possible for mobile clients to remain connected for that amount of time. Mobile clients
20 connect on an occasional basis, and when they do connect, the connection needs to move the smallest amount of data in the least amount of time. This is because wireless transports are not capable of moving large amounts of data quickly, and data is extremely expensive to move.

25 Existing client/server technologies based on persistent network connections were not designed to support occasional connections between low performance, low overhead handheld computing devices and existing enterprise computing systems. What is needed is a client/server architecture that supports occasional connections between low performance, low overhead mobile computing devices and existing enterprise computing systems. What is desired is an application development and deployment platform, such

2

that developers have the ability to create applications using a series of forms, tables, and communications agents, and the ability to deploy and maintain these applications. This platform should be implemented using an object model that can be easily ported to other hardware platforms and operating systems.

5 An architecture that allows multiple devices to connect concurrently to a single server is desired. This architecture should allow developers to connect any existing enterprise data source to handheld clients in the field. This architecture should allow developers to create two way links between any existing enterprise data source on a network, such as a database, mail server, or Internet news feed, and FormLogic client applications.

10 The improved client/server architecture should provide "transport independence", which is a unique requirement of field based applications. Sometimes it is necessary to connect over a serial cable, other times over a wireless local area network (LAN), and other times over the Internet. Such functionality has been addressed with "middleware" products. However, middleware products usually consist only of a series of "C" application programming interfaces (APIs) on client and server ends that require the developer to integrate them into an application. What is desired is to integrate "middleware" functionality directly into a specific server structure for which developers create "services".

15 Existing client/server APIs move a tremendous amount of data, such as Microsoft's ODBC (Open Database Connectivity). It is not feasible to use interfaces such as ODBC in the handheld or occasionally connected environment for two reasons. 1) The code size of ODBC is several megabytes - more than the entire memory of today's handheld devices and 2) ODBC is designed to work over a persistent connection with high bandwidth, such as Ethernet. Therefore, what is desired is a set of client/server APIs that can utilize a variety of transports to move a minimum amount of data over the wire or through the airways. To accommodate current and future transports, a message-based asynchronous communications protocol that is designed to work efficiently over low bandwidth, high latency networks is needed. This capability is required for evolving

wireless transports, such as these provided by the companies of ARDIS, RAM Mobile Data, and 2-way paging, such that developers will automatically be able to support them without making any changes to their applications.

Application software on a client device may not be the most recent available due to enhancements, fixes, and so forth. The architecture should support a users and groups model, wherein different applications modules can be distributed to a particular user or group. Using a version control for these applications components, users can automatically be updated with the latest version of an application upon connection.

Summary of the Invention

The client/server (C/S) architecture of the present invention is designed to allow the client to become a direct extension of the corporate data sources. The C/S components use an object management scheme and are preferably based on Microsoft's OLE technology. A 32 bit OLE control (OCX) is used to manage a connection with a multiple mobile personal digital assistant (PDA). This architecture allows the developer to manage a single connection with a single PDA device. It provides a completely asynchronous communications interface, providing multiple connections with multiple devices at the same time. Applications built with existing development tools can be enabled to either exchange data on demand, or provide facilities for a multi-port server allowing remote database access and e-mail access from the field. When used with client/server development tools such as Visual Basic, this server object allows developers to create direct connections between PDA devices, and nearly any host data source, including databases, mail servers, and Internet data sources.

In one aspect of the present invention there is a client/server system, comprising a portable client computer, comprising a client database, and a communications module; and a server computer, comprising a data storage, and a session module, in communication with the data storage, to non-persistently connect to the communications module and access the client database from time to time.

In another aspect of the present invention there is, in a computer network, including a server, a data storage, and a plurality of mobile clients, each mobile client having a database, a method of synchronizing each of the client databases and the data storage during a non-persistent connection, the method comprising (a) connecting one
5 of the mobile clients to a server having a session module; (b) manipulating the client database with the session module; (c) updating the data storage responsive to the manipulation by the session module; (d) disconnecting the client from the server; and (e) repeating (a)-(d) a plurality of times, each time with a different one of the mobile clients.

10 In another aspect of the present invention there is a client/server system, comprising a plurality of portable client computers, each client computer comprising a client database, and a communications module; a data storage; and a server computer persistently connected with the data storage, the server computer comprising a session
15 module to non-persistently connect to at least one of the communications modules and access the client database from time to time, wherein the plurality of portable client computers share data stored on the data storage.

In yet another aspect of the present invention there is a client/server system, comprising a plurality of portable client computers; a plurality of data storages; a server computer persistently connected to the plurality of data storages; and a plurality of
20 session modules executing on the server computer, each session module in data communication with one or more data storages that may be different than the one or more storages in communication with the other session modules.

Brief Description of the Drawings

25 Figure 1 is a block diagram of an exemplary known client/server system;

Figure 2 is a high-level block diagram of a preferred client/server embodiment of the present invention;

Figure 3 is a block diagram of the architecture of client components and server components of the system shown in Figure 2; and

Figures 4a and 4b are a diagram showing an exemplary client/server message exchange for a mail exchange session.

5

Detailed Description of the Preferred Embodiments

The following detailed description of the preferred embodiments presents a description of certain specific embodiments to assist in understanding the claims. However, the present invention can be embodied in a multitude of different ways as defined and covered by the claims. Reference is now made to the drawings wherein like numerals refer to like parts throughout.

10

The new FormLogic client/server (C/S) architecture is designed to allow a FormLogic client to become a direct extension of the corporate database. Previously, as described in applicant's patent, U.S. Patent No. 5,704,029, FormLogic clients provided data to host databases in the form of ASCII files that had to be imported into the target database. While reliable for batch file processing, this method did not provide a direct link between the client personal digital assistant (PDA) and the enterprise database. Furthermore, there was no way to automatically extract records from the enterprise database, and send them to the device. The new FormLogic C/S architecture overcomes these limitations by allowing developers to create direct links between PDAs and enterprise data sources using industry standard development tools.

15

20

The new FormLogic client/server components described herein use an object management scheme and are preferably based on Microsoft's OLE technology. A 32 bit OLE control (OCX) is used to manage a connection with a multiple PDA device. Because this OCX component is based on the industry standard component software model, it can be used with all leading industry standard development tools including Lotus Notes, Microsoft Visual Basic, Microsoft Access, Microsoft Visual FoxPro, Borland Delphi and PowerBuilder. The OLE implementation also provides the developer with a familiar object model and programming interface for integrating PDA technology into a predominantly Windows-based computing infrastructure. This allows developers to create

25

PDA-based solutions using their existing development tools, avoiding the need to develop for proprietary PDA operating systems.

A key component of the FormLogic client/server architecture is the FormLogic service object. The FormLogic service object allows developers to link PDA client applications for an unlimited number of user connections over a variety of transports without the need to worry about multi-user and concurrency issues. The service object allows the developers to write the application as if it were communicating with a single client, allowing them to focus on the application itself, rather than focus on communications transport, multi user, and concurrency issues.

10 The FormLogic service object has the following features:

- Ability to retrieve specific records based on a query;
- Ability to programmatically build and send records to the FormLogic Client;
- Ability to send asynchronous messages between the Clients and Server;
- Support for direct serial and modem connections;
- 15 - Support for AppleTalk (ADSP) and Internet (TCP/IP) network connections;
- OCX (ActiveX) implementation allowing integration with a host of development tools;
- Ability to customize the FormLogic Client Connection dialog during connections;
- MD5 authentication with the FormLogic Client;
- 20 - Complete Software Distribution interface allowing developers to programmatically install FormLogic forms, agents and tables during connections;
- 100% asynchronous interface.

A variety of applications are possible for the new FormLogic client/server architecture. Several exemplary applications are described below:

Provide Real-Time Access to Corporate Databases - Probably the most exciting use of the FormLogic C/S architecture is the integration of PDA technology with enterprise computing environments. Developers can now build applications that allow PDA devices to connect to virtually any type

of database, from Microsoft Access to Oracle, or even legacy systems. Once connected, developers can create applications that can query as well as update the database. For example, this allows for extremely fast development of field service applications, wherein field personnel connect to a remote database to retrieve work orders, and then later update the same work orders.

Create Robust E-mail Gateways - Using existing OLE controls from third parties, developers can easily integrate existing Messaging API-based or point-of-presence (POP)/Simple Mail Transfer Protocol (SMTP) mail systems. This provides opportunities for developers seeking to provide an application that provides access to both a host database and an enterprise e-mail system in a single connection.

Create Sophisticated Servers - Because the FormLogic Connection object is based on the OLE component technology, developers can create applications that host multiple simultaneous connections with a minimum amount of effort. Existing applications created with tools such as Microsoft Access, can quickly be turned into servers capable of hosting numerous simultaneous connections to PDA devices in the field.

Integrate PDA Data Transfer Functionality Into Existing Applications - Existing applications can easily be modified to provide simple data exchange facilities with PDA devices. This allows portions of databases to be carried into the field where they can be modified and later synchronized with the server database.

Referring to Figure 1, a typical client/server (C/S) system 100 previously known in software technology is shown. The system 100 includes a database 102, one or more servers 104, such as a mail server 104', and a local area network (LAN) 106. Alternatively, the LAN could be a wide area network (WAN) or an intranet. The database 102, the servers 104 and the LAN 106 collectively are known as the server portion 107 of

the client/server system. A plurality of clients, such as personal digital assistants (PDAs) 108, 110 and 112, are in communication with the server portion 107. The communication may be over a direct serial link, such as a serial cable or a modem.

In a traditional persistent connection based client/server model, clients remain connected and request data from the server as they need it. As the data is requested it is stored locally for manipulation and then discarded. The server still remains the primary and only main storage area for the data because the client always has access to it when needed.

Referring to Figure 2, a client/server system 130 of the present invention will be described. The client/server system 130 hereinafter may also be referred to as the FormLogic client/server system. The system 130 includes the database 102, the mail server 104', the LAN 106 and an administrator server 148. This portion 107' is similar to the server portion 107 of Figure 1. However, a FormLogic (FL) server 132 is connected to the LAN 106 in a persistent fashion to provide advantages not possible with the traditional system 100. The FL server 132 is connected to a plurality of client subsystems. For example, modems 134 and 134' interconnect the FL server 132 and PDA clients 136 and 136', respectively. An intranet or the Internet interconnects the FL server 132 and clients 142 and 142'. Client devices 146 and 146' are directly connected to the FL server 132 by a serial cable 147, for example.

Referring to Figure 3, the architecture of the FormLogic server 132 and a representative FormLogic client 136 will be described. The FL server and FL client were introduced in conjunction with Figure 2.

FL Client

The FL client 136 includes an FL Engine 160 which allows FormLogic applications to execute on a variety of handheld devices. The FL Engine 160 preferably runs on an Apple® MessagePad® Model 120 or Model 130 PDA using the Newton® version 2.0 operating system software. Of course, other portable computer devices and operating system software, such as Magic Cap from General Magic or Pegasus from

Microsoft Corporation, can be used in other embodiments. The FL Engine 160 is, in simple terms, a hardware independent virtual machine that allows a single application to work on various hardware platforms. A similar example is the Java virtual machine, licensed by Sun Microsystems, which may or may not execute within the context of a browser.

The FL client subsystem 136 preferably includes the FL Engine 160 comprising a user interface (UI) 162, a script engine 164, a communications module 166, and a data store 168. The user interface (UI) 162 and the script engine 164 have been previously described in applicant's copending patent application, U.S. Serial No. 08/247,777, which is hereby incorporated by reference. The communication module 166 packages data that is either being received or sent by the FL Engine 160 and handles interfacing the FL Engine to the FL Server 132 through the modem 134, the Internet 140 or the direct serial connection 147. Another embodiment may include a wireless LAN. The data store 168 includes one or more application programs 170 and a remote database 172 for storing the results of running the application program or storing data received from the FL Server 132, for example.

Because FormLogic clients, e.g., 136, do not maintain persistent connections with the FL server 132, they need to be able to store and access information while not connected to the host database, e.g., 182, or other data source. The FL Engine 160 incorporates a full local database implementation that allows data to be manipulated and collected by the FL client while not connected to the FL server 132. Upon connection, this local database 172 is automatically manipulated by the FL server 132. The FL server 132 can query the client database 172, add data to the client database, or remove data from the client database in order make updates to both the client and server databases to reflect changes that have happened on both sides since the last connection. Thus, a synchronization of the two databases is performed.

In the FormLogic C/S model, the FL server 132 maintains the primary enterprise database, e.g., 180, but instead of maintaining a full-time connection with clients, clients connect on an occasional basis. During this connection, the FL server 132 is responsible

for manipulation of the FL client database 172, including retrieving data that has been collected by the client since the last connection, or inserting new data in the database that has been added on the FL server 132 since the last connection. The client database 172 serves as a temporary representation of the host database, e.g., 180, because the client cannot maintain a full-time connection to the FL server 132. On the server side, a Remote Database API has been developed that allows developers to efficiently manipulate the client database 172 while sending a minimum amount of data over the connection.

FL Server

The implementation of the FormLogic Server architecture is unique. To allow the FL server 132 access to any data source a developer may already be working with, an API is provided between those existing data sources, e.g., 180, 182, and the FL server 132. The FL server 132 comprises an OCX (Microsoft OLE Custom Control), or software component, that can be embedded in a variety of existing development tools, including those tools that are already being used by developers to access enterprise data sources (e.g., MS Visual Basic, PowerBuilder, Delphi, Visual C++). This allows developers to easily extend the FL server 132 to their data sources using tools they are already familiar with.

The FL server 132 provides the ability to link hardware devices running the FL Engine 160 to access existing enterprise data sources on an occasional basis. It is optimized to communicate by exchanging a minimum amount of data, since the wireless transports are expensive and are characterized by high latency and low bandwidth. The FormLogic Server 132 serves as a "gateway" between FormLogic Clients (e.g., 136, 142, 146) and enterprise data sources (e.g., 180, 182). The server 132 supports what is known as a multi-tier client/server model in that it creates an intermediate server between the client and the "traditional" or "original" server.

The FL Builder (not shown) is a development tool, previously described in applicant's copending patent application, U.S. Serial No. 08/247,777, used to build FormLogic applications that can be executed on a variety of hardware platforms. It is

designed to give developers the look and feel of existing development tools, preferably Microsoft Visual Basic, while at the same time introducing some innovative features. It is a WYSIWYG tool that allows one to write code. It is unique in that it allows developers to create an object called a "communications agent" or just "agent" that encapsulates a communications "session".

Because mobile clients cannot maintain a persistent connection to the FL server 132, they must "connect" for short periods of time to perform a specified operation or set of operations. Each of these connections is referred to as a "session", during which time a specified set of operations are performed between the FL client and FL server. Examples of these sessions include connecting to retrieve work orders, checking inventory on a product, or retrieving a monthly price list update. Each "session" encompasses connecting the remote host, performing a specific task or set of tasks, and then disconnecting from the host. Because the connection times must be short, the FL client and FL server need to be able to perform the required tasks without user intervention. This is very different from a persistent connection based client/server model where the connection exists the entire time the application is used, and data is only retrieved when the user requests it.

Communications agents, also just known as "agents", are developed to describe the communications "session". Communications agents know how to connect to a particular host, perform a set of operations or tasks, which usually includes synchronizing the host data source, e.g., 180, with the client database 172, and then disconnecting. The idea is that a developer can create a communications agent that represents each of the communications sessions that a field user may need. For example, there may be a communications agent that retrieves work orders, updates work orders, or downloads a price list. There may also be a communications agent that simply checks inventory on a particular product. In general, communications agents are designed to encompass the fundamental operations that are needed to exchange data between a client and a host for a particular application.

The agent implementation is simple, and utilizes a simple software "object" to describe the agent. The developer creates a named object and provides a name, as well as

other properties, which, upon connection, tell the FL server what type of session the FL client is requesting, as well as any parameters required to perform specific operations in that session. Agents may also specify a particular transport to minimize the cost of a connection, e.g., an agent needing a long connection time would use a less expensive type of transport.

An exemplary Session1 200 called Daily Connect includes three tasks: Task1 204, e.g., GetMail; Task2 206, e.g., SendMail; and Task3 208, e.g., UpdateInventory. Another exemplary Session2 202 includes two tasks: Task4 210, e.g., InterrogateInventory; and Task2 206, SendMail.

The FL Server 132 includes a message handler 184 for interfacing the FL Server 132 to the FL Engine 160 through the modem 134, the Internet 140 or the direct serial connection 147, for example. The message handler 184 communicates with each instantiation of the FormLogic Connection object. For example, as shown in Figure 3, a Connection object 194 may be associated with Client A 136 and a Connection object 196 may be associated with Client C 146. Each of these connections are independent, and a plurality of connections may be concurrent. Each Connection object has a current task pointer for pointing to the current task. When the task is completed, the pointer is incremented to point to the next task in the session. Each of the Connection objects is pointing to a particular task in a particular session. In Figure 3, for example, Connection object 194 is pointing to Session1 and Connection object 196 is pointing to Session2. There is one real object for each session, and each connection points to its current place (task) in the session.

A set of tasks are provided or handled by a service. A service defines the relationship between a client application and an enterprise data source. Examples of services include Mail, World Wide Web Gateway, or Inventory. For example, a Mail service 190 is a service that provides the GetMail task 204 and SendMail task 206, and is connected to the data source 180. An Inventory service 192 provides the UpdateInventory task 208 and InterrogateInventory task 210, and is connected to an inventory data source 182, for example. Each Connection object has its own instantiation of the service class

associated with it. For example, Task1 and Task2 comprise an instantiation of Service1 for Connection object 194. The service is written as if the server was in communication with a single client. Multiple copies of the service are needed for each single client connection coming into the FL server. Each instantiation of the service does not maintain connection to its data source; only the "master" service object maintains connection to the associated data source. The service instantiations can be considered as interfaces between the "master" service and the connection.

FL Server APIs

The FL Server APIs allow developers to write services for FL Server that will link FormLogic client applications to existing enterprise data sources without worrying about multi-user and concurrency issues. The FL Server APIs are used from within Visual Basic or other development tools to communicate with FL Engine and allow server applications for an unlimited number of user connections over a variety of transports without the need to worry about multi-user and concurrency issues. The server object allows the developers to write the application as if it were communicating with a single client, allowing them to focus on writing services for the application itself, rather than focus on communications transport, multi-user, and concurrency issues.

Service methods are invoked by services, usually with the convention "connobj.Method()". Some methods such as of the remote database APIs have corresponding events that are triggered by the messages from the client indicating the results of the actions invoked by the method. The Service APIs fall into three distinct categories: the Remote Database APIs, the Messaging APIs, and Utility APIs:

Remote Database APIs

These calls are used to directly manipulate the client database during a connection. When invoking Remote Database APIs from services, corresponding events will be passed back to the services that generated the call. A remote procedure call mechanism is used. There is a corresponding event for every Remote Database API.

- 5 Corresponding events are guaranteed to be called, assuming the method used to trigger it did not return an error. Methods and Events for the Remote Database APIs are listed in Table 1 below.

TABLE 1

Method	Corresponding Event
CreateRecordSet (RSName, tableName, queryDef)	OnCreateRecordSet
GetNextRecord (RSName)	OnGetNextRecord
AddRecord (RSName, FLRec)	OnAddRecord
DeleteRecord (RSName, FLRec)	OnDeleteRecord
DeleteRecordSet (RSName)	OnDeleteRecordSet
Event	
OnCreateRecordSet (errcode, RSName, recordCount)	
OnGetNextRecord (errcode, RSName, record)	
OnAddRecord (errcode, RSName)	
OnDeleteRecord (errcode, RSName)	
OnDeleteRecordSet(errcode, RSName)	

- 10 Method details and Event details for the Remote Database APIs are listed in Table 2 below.

TABLE 2

CreateRecordSet(tableName, queryDef, RSName)

	<i>RSName</i>	String	User defined recordSet name.
	<i>tableName</i>	String	Name of table on FormLogic Client.
	<i>queryDef</i>	String	SQL "Where" clause
5	returns errcode		

GetNextRecord(*RSName*)

	<i>RSName</i>	String	User defined recordSet name.
10	returns errcode		

AddRecord(*RSName*, *FLRec*)

	<i>RSName</i>	String	User defined recordSet name.
15	<i>FLRec</i>	FL Record Object	Record to send to FormLogic Client.
	returns errcode		

DeleteRecord(*RSName*, *FLRec*)

	<i>RSName</i>	String	User defined recordSet name.
20	<i>FLRec</i>	FL Record Object	Record to send to FormLogic Client.
	returns errcode		

OnCreateRecordSet(errcode, *RSName*, recordCount)

	<i>errcode</i>	Long	Error code.
25	<i>RSName</i>	String	User defined recordSet name

<i>recordCount</i>	Long	Number of records in RSName recordSet
--------------------	------	---------------------------------------

OnGetNextRecord(errcode, RSName, FLRec)

	<i>errcode</i>	Long	Error code.
5	<i>RSName</i>	String	User defined recordSet name
	<i>FLRec</i>	FL Record Object	Record received from FormLogic Client.

OnAddRecord(errcode, RSName)

	<i>errcode</i>	Long	Error code.
10	<i>RSName</i>	String	User defined recordSet name

OnDeleteRecord(errcode, RSName)

	<i>errcode</i>	Long	Error code.
	<i>RSName</i>	String	User defined recordSet name

15

Messaging APIs

20 These APIs are used to send specific messages to FormLogic Agents 170 (Figure 3) on the client device. A rules-based specification of a particular agent on the client can be done. Messages can be used to send any type of data in real time, and allows the agent on the client side to decide how to handle it. Virtual sessions can be established with these APIs. Methods and Events for the Messaging APIs are listed in Table 3 below.

TABLE 3

T0180

5	Method
	Send
	Reply
	Event
	OnMessage

Method details and Event details for the Messaging APIs are listed in Table 4

10 below.

TABLE 4

T0181

15

Send(agentID, methID, message)			
agentID	Long	Client agent ID	
methID	Long	Developer defined method ID	
message	FL Record Object	Record to send to FormLogic Client.	
returns errcode			

20

Reply(agentID, methID, message)			
agentID	Long	Client agent ID	
methID	Long	Developer defined method ID	
message	FL Record Object	Record to receive from FormLogic Client.	
returns errcode			

25

OnMessage(errcode, objID, methID, FLRec)			
errcode	Long	Error code.	
objID	Long	User defined object ID	
methID	String	User defined method ID	
FLRec	FL Record Object	Record to send to FormLogic Client.	

30

18

Utility APIs

Utility APIs don't actually send any data between the server and the client. They are used to perform functions such as setting timers, writing to the system log, and controlling the client's connection dialog. Methods and Events for the Utility APIs are listed in Table 5 below.

TABLE 5

Method

SetTimer (*Name*, *interval*)

LogEntry (*messagetype*, *message*)

SetStatus (*message*, *gaugeType*, *currVal*)

GetUsername ()

GetAgentParm (*parmName*)

StartNextTask

Event

OnTimer (*errcode*, *RSName*)

Method details for the Utility APIs are listed in Table 6 below.

TABLE 6

LogEntry(*messagetype*, *message*)

<i>messagetype</i>	Integer	Message type
<i>message</i>	String	Text message
returns errcode		

SetStatus(*message*, *gaugeType*, *currVal*)

<i>message</i>	String	Text message
<i>gaugeType</i>	Integer	kProgressbar = 0, kBarberPole = 1
<i>currVal</i>	Integer	Gauge position between 0 and 100.
returns errcode		

The *LogEntry* method provides the ability to write to the system log when a selected activity occurs, e.g., when the user logs on or off. Error messages can be written to the system log specific to a particular service that the developer is writing. The system log can be read by the system administrator. This method could be used for
5 billing clients. For example, a log entry could be made every time a message is read from a news feed service.

The *SetStatus* method gives the FL server a means to update a dialog box on the client side without causing an extra message to be sent over the link, i.e., this method does not generate additional traffic on the link. The update message is jammed into the
10 next message or packet that is being sent.

The *SetTimer* method provides a way to determine if the client has responded. The developer can set an alarm time interval and an alarm timer name to measure elapsed time. After the alarm is triggered, an OnTimer Event is fired with the name of the alarm timer. This method is used to prevent code block or lockup, and is used in
15 place of a timeout because different transports take different amounts of time to respond.

The *GetUsername* method can be used inside the service to obtain the name of a logged-on user.

The *GetAgentParm* method allows a service to extract any of the parameters sent
20 over to the FL server at log-on time. For example, a Mail agent may pass over a log-on name, a password and a mail server ID. The FL server stores these parameters for the developer to use inside the service. In the current example, the developer could use the internet protocol (IP) address of the mail server inside the service.

The *StartNextTask* method is used by the developer to execute the next task after
25 the current task is completed. After StartNextTask is called, the FL server takes over and automatically gets and executes the next task in the session.

Example Client/Server Message Exchange

Referring to Figures 4a and 4b, an exemplary client/server message exchange for
30 a Mail Exchange session 230 will be described. The exchange between the FL client

136 and the FL server 132 is shown in a graphical time dependent format. Time increases while traversing downward on the graph.

The session shown in Figures 4a and 4b illustrates a single service. Of course, other services could be part of a single session. The session shown in Figures 4a and 4b shows Task1 (204) and Task2 (206) of Session1 (200). Task3 (208) is not shown in this session.

The session 230 begins with a user initiating a login 232. A message 234 passes a username of the user, the user's password, an application profile, and a session name to the server. The application profile is a client list of all its applications and includes forms, agents, tables and respective version numbers. The server maintains an administration profile which is a list of the most current applications and the version numbers of the applications and their forms, agents and tables.

The server authenticates 236 the received data from the message 234, i.e., the received username and password are verified as correct. In response, the server sends the time of authentication 238 to the client. The server also checks the application profile received from the client against its administration profile to determine if the client applications are current as determined from the version numbers. If any applications are not current, a synchronize software operation 240 is initiated by the server to update the client machine. The most current application(s) are then sent over to the client using a handshaking mechanism. Since FL application and updates are relatively small, this process should complete rather quickly.

At the completion of the software synchronization 240, the server accesses the exemplary ExchangeMail session. StartNextTask 250 is automatically called by the server for the first task (GetMail) in the session. A CreateRecordSet method 252 (Remote Database API) is invoked by the server. A recordSet object represents a plurality of records in a base table or the records that result from running a query. In this instance, the name of the recordSet is "Mail" and the "_direction='outgoing'" string is a query that identifies records in the Mail table for which the direction is outgoing. The communication module 166 at the client creates a set of records on the client device

in response to the message 252. It also sends an OnCreateRecordSet event 254 to the server. The recordSet name ("Mail") and a recordCount, which is the number of records in the recordSet, e.g., two pieces of outgoing mail, are returned to the server.

5 The server then utilizes the GetNextRecord method 256 to retrieve the first record from the "Mail" recordSet. The communication module 166 at the client responds with an event which is received by OnGetNextRecord 258 at the server. The recordSet name ("Mail") and a FL record object (the Record), are returned to the server. The FL record object is an object of type FL record that encapsulates a set of fields and their values. In this instance the results are sent to mail server, but could also be sent to
10 a printer, a screen display, or to a database, for example. In the preferred embodiment, after the record is received at the server, the record is deleted at the client. In another embodiment, the record is maintained at the client. The second outgoing mail record is retrieved by a second GetNextRecord method 260 and OnGetNextRecord event 262. At the completion of event 262, both pieces of mail have been retrieved from the client by
15 the server.

Continuing the session 230 on Figure 4b, the session calls StartNextTask 270 to invoke a SendMail task. The server utilizes an AddRecord method 272 to send mail to the client. The recordSet name ("Mail") and the FL record object (the Record), are passed to the client. The mail server or other source of data is interrogated to determine
20 the data to be sent to the client. This example assumes that the recordSet is available from the previous GetMail task and that the SendMail task is done after GetMail. Alternatively, a CreateRecordSet method could be invoked at the beginning of the SendMail task (before AddRecord). Using a flag or other indicator, a check could be done to determine if the recordSet is already created, and if not, the CreateRecordSet
25 method would be invoked.

The client receives the FL record object (the Record) and responds to the AddRecord method with an OnAddRecord event 274. The server checks every event to determine if an error code is included, and if so, takes appropriate action. Every method also returns an error code that is also checked. A second AddRecord method 276 and a

second OnAddRecord event 278 are performed for a second FL record object sent to the client.

5 A SetStatus method (not shown) of the Utility APIs is called after each
GetNextRecord method and each AddRecord method to update a progress thermometer
in a dialog, for example. At the completion of the OnAddRecord event 278, the server
calls a Send method 280 to send a dialog message in a dialog box on the video screen of
the client, such as "2 mail records sent, 2 mail records received". The server 132 then
invokes a disconnect task 290 by a Disconnect method 292. In the presently preferred
embodiment, the client disconnects and responds with a OnDisconnect event 294 to the
10 server. In another embodiment, the client does not respond with the OnDisconnect
event 294, but does perform the disconnect housekeeping task.

15 While the above detailed description has shown, described, and pointed out the
fundamental novel features of the invention as applied to various embodiments, it will
be understood that various omissions and substitutions and changes in the form and
details of the system illustrated may be made by those skilled in the art, without
departing from the spirit of the invention.